Relieving Capacity Limits on FPGA-Based SAT-Solvers

Leopold Haller
Computing Laboratory
Oxford University, United Kingdom
leopold.haller@comlab.ox.ac.uk

Satnam Singh
Microsoft Research
Cambridge, CB3 0FB, United Kingdom
satnams@microsoft.com

***Abstract*—FPGA-based SAT solvers have the potential to dramatically accelerate SAT solving by effectively exploiting fine-grained pipeline parallelism in a manner which is not achievable with regular processors. Previous hardware-based approaches have relied on on-chip memory resources to store data which, similar to a CPU cache, are very fast, but are also very limited in size. For hardware-based SAT approaches to scale to real-world instances, it is necessary to utilise large amounts of off-chip memory. We present novel techniques for storing and retrieving SAT clauses using a custom multi-port memory interface to off-chip DRAM which is connected to a processor core implemented on a medium sized FPGA on the BEE3 system. Since DRAM is slower than on-chip memory resources, the parallelisation which can be achieved is limited by memory throughput. We present the design and implementation of a new parallel architecture that tackles this problem and estimate the performance of our approach with memory benchmarks.**

I. Introduction

SAT solvers have been established as popular black-box reasoning techniques in a number of application areas, most notably, formal verification of hardware and software. This can be partially attributed to the fast rise in solving efficiency over the last 15 years. One possibility of increasing solving efficiency further is to make use of the fine-grained parallelism that is offered by hardware platforms. Previous approaches have relied on on-chip memory resources which are fast and allow for parallelised access, but impose strict limits on the size of input instances.

In this paper, we explore the feasibility of building a hardware-based SAT solver that directly accesses off-chip DRAM memory resources. This has the advantage that the size of SAT instances solved by our hardware solver are orders of magnitude larger than what is possible when storing instance data using only on-chip memory. The disadvantage is that it creates a memory bottleneck due to the memory access characteristics of DRAM. We present an implementation of a Boolean constraint propagation (BCP) unit on the BEE3 multi-FPGA board.

Our design uses novel techniques for clause retrieval and propagation that utilise fine-grained parallelism in spite of this bottleneck. For the clause retrieval step, we adapt the BCP algorithm to independently access multiple memory channels. The unique advantage of our approach is that it does not impose the strict instance size limits that are common with other hardware-based SAT solvers. The evaluation of our implementation is work in progress. We present initial memory benchmarks to estimate the feasibility of our approach.

II. Related Work

A survey of techniques published until 2004 is given in [1]. Early work on reconfigurable hardware SAT focuses on *instance specific* approaches (e.g., [2], [3], [4], [5]), in which a circuit is generated specific to a single SAT instance. This requires computationally expensive circuit resynthesis and reconfiguration of the hardware once a new instance is to be evaluated and severely limits the size of possible input instances. *Application specific* hardware solvers do not require reconfiguration between solving instances. A popular approach is to implement BCP, the most work intensive step of the popular DPLL procedure, on hardware, and handle more complex tasks such as conflict analysis and decision heuristics in software [6], [7], [8], [9]. Fully functional hardware solvers are presented in [10], [11], [12].

Capacity is an issue for all these solvers. Examples of more large-scale approaches include the BCP accelerator presented in [9], which can accommodate 64K variables and equally many clauses of length 9, or the solver in [12], which can accommodate 10K variables and 280K fixed-length clauses. Many SAT instances of practical interest are not representable within these restrictions.

A number of methods have been proposed to increase the capacity of hardware-based solvers: Examples include using a larger FPGA [4] or multiple FPGAs [7], [8], [9], splitting the problem into small subproblems [5], partitioning the instance into small-sized frames that are loaded on-demand [12], or combining a software solver with a hardware solver for small-size subproblems [10]. Our approach, in contrast, explores the feasibility of directly accessing off-chip memory resources.

III. Memory Access Patterns in SAT Solvers

In FPGA designs, very small amounts of data can be stored on arrays of state-holding flip-flops. Larger amounts can be stored in dedicated Block RAM (BRAM) modules on the FPGA chip, or off-chip on external RAM. On-chip memory is very limited, with typical sizes smaller than 4MB, but access is fast and can be performed in parallel. Access to DRAM

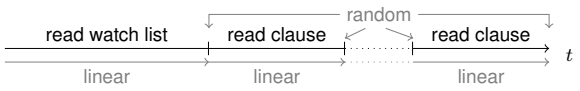

Fig. 1. Watch list and clause read operations in Algorithm 1

	PC (400MHz RAM)	Virtex5 (250MHz RAM)
rnd./lin.	93.7 / 1066.7	44.4 / 1066.7
quasi-lin.	691.9	984.6

TABLE I
AVERAGE MEMORY ACCESS SPEED (MB/S)

memory is performed via an external memory controller using an asynchronous protocol. We have used a freely available controller presented in [13] for our implementation. The time required for a single read and write command depends on a number of factors including access locality, memory clock speed and the implementation of the memory controller. Random accesses are, on average, significantly slower than linear streaming access.

Most modern SAT solvers are based on the Conflict Driven Clause Learning (CDCL) framework, which utilizes clause learning and backjumping ([14], [15]), and spend most of the runtime in Boolean Constraint Propagation (BCP). BCP can be efficiently implemented using a watched-literal scheme [16] where two literals in each clause are observed for changes. Literal watching can be implemented by associating each literal with a list, which records the clauses that have to be visited when the literal is contradicted during search.

Algorithm 1 The BCP step

```
1: procedure Propagate(l : literal)
2:     wl ← readWatchList(l)                          ▷ R
3:     while HasNextClause(wl) do
4:         c ← readNextClause(wl)                     ▷ R
5:         v ← readVariableValues(c)                  ▷ R
6:         s ← analyse(v, c)
7:         if s = Conflict then return Conflict
8:         else if s = Deduction then writeDeducedValue()   ▷ W
9:         changeWatchLits()                          ▷ W
10:    return Unknown
```

In Algorithm 1, the inner core of the BCP algorithm is presented in a way that emphasizes memory accesses. ReadVariableValues fetches the values assigned to variables occurring in a clause. The Analyse function determines the status of a clause and returns a result that indicates if an action needs to be taken. Finally, ChangeWatchLits modifies watch lists in accordance with the two-watched literal scheme.

The memory access pattern for reads is illustrated in Figure 1. Reads are issued in a linear fashion on successive addresses, with intermittent single random accesses. We will refer to this access pattern as *quasi-linear*.

In order to estimate the viability of using DRAM in a reconfigurable-hardware SAT solver, we compared the efficiency of quasi-linear memory accesses on the BEE3 platform with the same access pattern implemented in software and run on an average PC. Since modern CPUs have large multi-level caches which allow fast access to recently used data, it is not a priori clear that the performance is similar, even when similar types of main memory are used.

For our experiments and implementation of the BCP core, we use a pre-production version of the BEE3 FPGA board which has four XC5VLX110T FPGAs. Each FPGA has two independent memory channels connected to dual channel DDR2-533 RDIMMs with each channel populated with 8GB of memory (giving a total of 64GB for the whole system). The FPGAs are connected in a ring and a cross-over board also provides direct connections between the other two FPGAs. Although this platform was primarily developed for the emulation of multi-core processors we believe it is an interesting platform for hardware SAT-solving because of the large amount of off-chip memory and the ability to use eight independent memory channels to experiment with hardware parallelisation techniques.

The BEE3 system also provides a variety of I/O ports including for each FPGA an RS232 serial port, dual 10GBase-CX4 Ethernet interfaces, a single PCI-Express x8 end-point slot and a Gigabit Ethernet port. We use the Gigabit Ethernet port to communicate with a host PC running Windows 7.

We compared DDR2 memory access speed on a 3GHz CPU with memory clocked at 400MHz and a Virtex-5 FPGA with memory clocked at 250MHz. Random address values were precomputed and read (linearly) from an array in the software case, and generated on-the-fly in the hardware case. The memory controller has a granularity of 256 bit per memory access (288-bits including error correction bits). Hence, for the linear and quasi-linear access cases, a single read and write operation can manipulate 8 integers of width 32 at once.

The test setup consisted of reading and incrementing 256MB of 32 bit integers. The results are shown in Table I, which shows read/write speeds for a completely random, linear and quasi-linear access patterns. The quasi-linear access pattern reads 64 words linearly before performing a random address jump. As can be seen from the table, the access speed for quasi-linear access is comparable on the two platforms, despite the lower memory clock speed of the FPGA. Since quasi-linear accesses are characteristic for the DPLL algorithm, this result gives some preliminary indication that a hardware-based implementation of DPLL with direct DRAM access is feasible.

IV. BUILDING A DRAM-BASED BCP-CORE

Since straight-forward highly parallel approaches are not practicable when accessing DRAM directly, we base our implementation on modern software CDCL solvers and enhance it with fine-grained parallelism where possible. The only data which we store on-chip are the current value of variables, all other data is kept in off-chip DRAM. In this situation, it becomes necessary to explore parallelisation techniques that are still viable in the context of the memory bottleneck that is created by off-chip data storage.

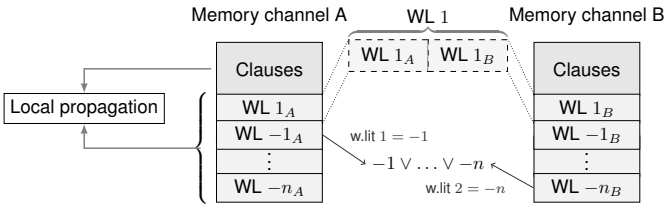

Fig. 2. Parallel watched literal scheme

A. The Parallel Watched-Literal Scheme

While DRAM access is inherently sequential, the BEE3 board retains some options for concurrent off-chip memory operations by offering multiple RAM channels, each of which is connected to its own RAM chip and can be controlled independently.

A key aspect of our approach is the ability to exploit two independent memory channels on each of the FPGAs, since it maps naturally to the two-watched literal scheme. In the watched literal scheme, two literals of each clause are designated and watched for changes. This is implemented by keeping a list for each literal, and appending all clauses to it in which it is being watched.

In our BCP implementation, we parallelise the two-watched-literal scheme by watching each of the two literals of a single clause on a separate memory channel (see Figure 2). Each literal is associated with two watch lists that are stored on separate channels A and B. Clause data is stored redundantly on the two memory chips. This allows to localise the inner core of BCP to require only memory accesses on a single memory channel. When the routine in Algorithm 1 is executed, the two partial watch lists are fetched independently on the two memory chips. After this step, the while-loop at line 3 of the algorithm can be executed completely in parallel by performing propagation local to data stored on each of the memory channels.

Redundant storage of clause data creates a memory overhead that is not significant in view of the large amount of available off-chip memory, but can speed up the processing of a watch list by up to 100%. By dividing the watched literals between the two memory channels, the average length of watch lists on each channel will be equal.

B. Parallel Inference

When relying on off-chip memory resources, clauses need to be read sequentially after the watch list is retrieved. The amount of possible parallelisation in analysing clauses is directly limited by the rate at which clauses can be streamed from memory.

After a clause has been retrieved, its variables' values have to be read (line 5 in Algorithm 1). The actual analysis step (line 6) can then be performed in a single clock cycle by a dedicated analysis circuit. We store variable values on on-chip memory resources. A value can therefore be accessed in a single cycle. Large clauses might still require a number of cycles to fill up all variable values of interest. Depending on how fast a clause can be streamed from memory and how many variable values need to be fetched before the status of a clause can be determined, there can be an overlap with new clauses arriving while a previous clause is still fetching variable values, as illustrated in Figure 3.

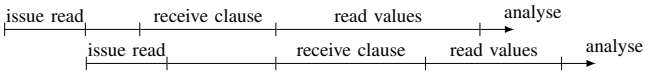

Fig. 3. Timing of the clause analysis step

To speed up these cases, we have implemented a limited form of inference parallelism. The clause analysis step is performed by propagator cores, which are assigned clauses that arrive from memory. Once a core has received a clause, it starts issuing requests for variable values to a common bus, and listening for useful variable values on another bus. Once enough variable values have been received to determine clause status, the core sets a ready flag and waits for the next clause assignment. In most cases, a core does not need to fetch all variable values in order to determine the status of a clause. In case a clause is either satisfied or is neither conflicting nor leads to a deduction, the result can be determined early.

The number of propagator cores is a parameter in our design. Once the analysis speed outpaces clause throughput no further efficiency gains can be obtained by adding cores. In our implementation, we have therefore instantiated the design with two propagation cores per memory channel.

C. Algorithmic Description of the BCP step

We will now give an algorithmic description of our solver, before discussing the implementation architecture. We present an overview in Algorithm 2. The procedures BCP and BCPCore correspond to the hardware modules of the same name that are discussed in the next section.

Algorithm 2 Algorithmic description of hardware BCP

```
1: procedure BCP(l : literal)
2:     q ← {l}
3:     while |q| > 0 do
4:         p ← pop(q)                                   ▷ pop queue
5:         BCPCore(p, A), BCPCore(p, B)                 ▷ execute in parallel
6:         if conflict(A) ∨ conflict(B) then return Conflict
7:         append(q, deductions(A) ∪ deductions(B))
8:
9: procedure BCPCore(l : literal, X : memory channel)
10:    wl ← issueReadWatchList(l, X)                    ▷ watch list fetch
11:    while ¬watchListReceived() do
12:        addr ← waitForClauseAddress();
13:        issueClauseRead(addr,X)                      ▷ clause fetch
14:    while ¬allClausesReceived(wl) do                 ▷ clause propagation
15:        c ← waitForClause(); assignToFreePropagationCore(c)
16:    writeBackWL(l,X)                                 ▷ write new watch list for l
17:    appendWatches(X)                                 ▷ append changed watched literals
```

In the BCP step, propagation literals are incrementally taken from a queue, after which BCPCore is executed in parallel on memory channels A and B. Each BCPCore reads its (partial) watch list from memory and issues "clause read" commands as soon as clause addresses are received. Arriving clause data

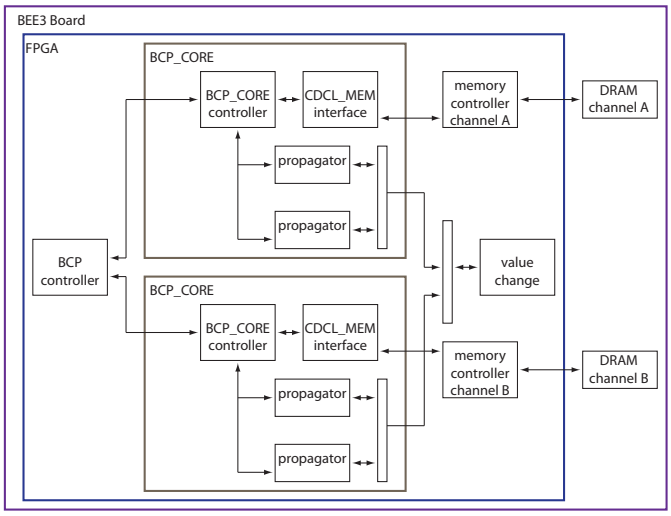

Fig. 4. Implementation Architecture

is distributed to the propagator cores which record their results for later evaluation. The addresses of those clauses which remain in the watch list (e.g., already satisfied clauses) are written back to memory in the call to WriteBackWL. The addresses of all other clauses are appended to their new watch lists in the AppendWatches step. Deduction results are recorded and processed in the main BCP procedure. If no conflict is found, the procedure appends new deduction results to the BCP queue.

D. Implementation Description

In our implementation, we use external DRAM to store clause and watch list information, while we use on-chip BRAM to store variable values. We use an openly available DDR2 memory controller [13].

The architecture of our BCP module is presented in Figure 4. The "BCP controller" block corresponds to the BCP procedure in Algorithm 2. It manages a BCP queue, issues propagate commands to the two "BCP_CORE" modules, and controls the modification of watch lists. The two BCP cores receive propagation literals from the BCP controller, issue memory requests to the "CDCL_MEM interface" module and distribute clauses on their partial watch list between free "propagator" cores. The propagator units access a common bus to read literal values.

In our BEE3 implementation, we limit the clause size to 24 literals to enable efficient propagation, and impose a limit on total size of watch lists to 256 clause addresses (128 per memory channel). This allows storage of instances with up to 1 million variables and 70 million clauses. We have validated our approach in simulation and synthesized our circuit with a memory clock frequency of 200 MHz and control logic clocked at 100 MHz. Obtaining benchmark results is work in progress.

V. Conclusion

In this paper we have presented an implementation of a Boolean constraint propagation core that does not rely on limited on-chip memory resources to store instance data, but instead directly accesses off-chip DRAM. Based on the memory access behaviour of CDCL solvers and the characteristics of DRAM, we have proposed techniques that introduce parallelism in spite of the memory bottleneck created by using off-chip resources. The evaluation of our implementation is work in progress. Our initial exploration is encouraging and we conclude that there is a good potential for implementing high performance parallel hardware SAT solvers by carefully designing and tuning the circuits that make up the memory hierarchy.

Future work includes the completion of the system which drives the parallel hardware BCP core by adapting an existing SAT-solver like MiniSAT and executing it on an embedded soft processor on the Virtex-5 FPGA or on an embedded hard core processor like a PowerPC or ARM core. Currently we use just one of the four FPGAs on the BEE3 system and in future work we hope to exploit all four FPGAs.

References

[1] I. Skliarova and A. B. Ferrari, "Reconfigurable hardware SAT solvers: A survey of systems," *IEEE Trans. Computers*, vol. 53, no. 11, pp. 1449–1461, 2004.

[2] P. Zhong, M. Martonosi, P. Ashar, and S. Malik, "Using configurable computing to accelerate Boolean satisfiability," *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 18, no. 6, pp. 861–868, 1999.

[3] T. Suyama, M. Yokoo, H. Sawada, and A. Nagoya, "Solving satisfiability problems using reconfigurable computing," *IEEE Trans. VLSI Syst.*, vol. 9, no. 1, pp. 109–116, 2001.

[4] M. Platzner and G. D. Micheli, "Acceleration of satisfiability algorithms by reconfigurable hardware," in *FPL*, ser. Lecture Notes in Computer Science, R. W. Hartenstein and A. Keevallik, Eds., vol. 1482. Springer, 1998, pp. 69–78.

[5] M. Abramovici and J. T. de Sousa, "A SAT solver using reconfigurable hardware and virtual logic," *J. Autom. Reasoning*, vol. 24, no. 1/2, pp. 5–36, 2000.

[6] J. de Sousa, J. P. Marques-Silva, and M. Abramovici, "A configware/software approach to SAT solving," in *FCCM*, 2001.

[7] A. Dandalis and V. K. Prasanna, "Run-time performance optimization of an fpga-based deduction engine for SAT solvers," *ACM Trans. Design Automation of Electronic Systems*, vol. 7, no. 4, pp. 547–562, Oct. 2002.

[8] J. D. Davis, Z. Tan, F. Yu, and L. Zhang, "Designing an efficient hardware implication accelerator for SAT solving," in *SAT*, ser. Lecture Notes in Computer Science, H. K. Büning and X. Zhao, Eds., vol. 4996. Springer, 2008, pp. 48–62.

[9] ——, "A practical reconfigurable hardware accelerator for boolean satisfiability solvers," in *DAC*, L. Fix, Ed. ACM, 2008, pp. 780–785.

[10] I. Skliarova and A. B. Ferrari, "A software/reconfigurable hardware sat solver," *IEEE Trans. VLSI Syst.*, vol. 12, no. 4, pp. 408–419, Apr. 2004.

[11] M. Waghmode, K. Gulati, S. P. Khatri, and W. Shi, "An efficient, scalable hardware engine for Boolean satisfiability," in *ICCD*. IEEE, 2006.

[12] K. Gulati, S. Paul, S. P. Khatri, S. Patil, and A. Jas, "Fpga-based hardware acceleration for boolean satisfiability," *ACM Trans. Design Autom. Electr. Syst.*, vol. 14, no. 2, 2009.

[13] C. Thacker, "DDR2 controller for the BEE3," http://research.microsoft.com/en-us/downloads/12e67e9a-f130-4fd3-9bbd-f9e448cd6775.

[14] R. J. Bayardo and R. Schrag, "Using CSP look-back techniques to solve real-world SAT instances," in *AAAI/IAAI*, 1997, pp. 203–208.

[15] J. P. Marques-Silva and K. A. Sakallah, "GRASP - a new search algorithm for satisfiability," in *ICCAD*, 1996, pp. 220–227.

[16] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient sat solver," in *DAC*. ACM, 2001, pp. 530–535.